Lecture 31-33 - Rootfinding

Table of Contents

31.1 Why is it important to be able to find roots? 1
31.2 Iterative Solutions and Convergence 1
31.3 Successive Approximations 2
    Code for Successive Approximations in Matlab 5
31.4 Bisection Method of Rootfinding 6
    Code for Bisection Method in Matlab 8
31.5 Single Variable Newton-Raphson Method 9
    Code for Newton-Raphson Method in Matlab 11
31.6 Single Variable Newton-Raphson - Numerical Approximation to Derivatives 12
    Code for Newton-Raphson Method w/ Numerical Derivatives in Matlab 14
31.7 MATLAB fzero.m - Search and Interpolate 15
31.8 How Many Roots do nonlinear equations have and how do we find them? 16
31.9 Single Variable Rootfinding - MATLAB - rootfinder.m 21
31.10 Single Variable Rootfinding - MATLAB - GUIs 23

Text: Supplementary notes from Instructor

31.1 Why is it important to be able to find roots?

Chemical Engineering is a methodical practice. With the computational tools available today, the biggest difficulty in solving a problem should be properly posing the problem. The actual computations required to extract numeric answers should be trivial. There should be no blood, sweat, or tears required in the numerical solution. Perhaps, for many of you, this is not yet the case. At the end of the course, this should be the case. If you can properly pose the problem, you can solve it.

This section of the course discusses the theoretical basis of some of these numerical techniques. We then apply the techniques to problems typically presented to the chemical engineer. Then we look at how to use a modern computational tool to solve them.

Numerical methods for root-finding have undergone a lot of scrutiny and development with the advent of powerful computers. Computer scientists and mathematicians have teamed up to development powerful algorithms for efficient and accurate root-finding techniques. It is beyond the scope of this course to delve into these state-of-the-art algorithms, used by such software as MATLAB. However, we will discuss some of the basic algorithms, so you understand the fundamental concepts at work.

31.2 Iterative Solutions and Convergence

Our goal is to find the value of x that satisfies the following equation.

\[ f(x) = 0 \]  

(31.1)

where \( f(x) \) is some nonlinear algebraic equation. The value of \( x \) that satisfies \( f(x)=0 \) is called a root of \( f(x) \).
Therefore, the procedure used to find \( x \) is called root-finding.

An example of a nonlinear algebraic equation is

\[ f(x) = x - \exp(-x) = 0 \]
All root-finding techniques are iterative, meaning we make a guess and we keep updating our guess based on the
value of \( f(x) \). We stop updating our guess when we meet some specified convergence criterion. The convergence
criteria can be on \( x \) or on \( f(x) \).

\[
\text{err}_{f,i} = |f(x_i)|
\]  (31.2)

Since \( f(x) \) goes to zero at the root, the absolute value of \( f(x) \) is an indication of how close we are to the root.

Alternatively we can specify a convergence criteria on \( x \) itself. In this case, there are two types of
convergence criteria, absolute and relative. The absolute error is given by

\[
\text{err}_{x,i} = |x_i - x_{i-1}|
\]  (31.3)

This error has units of \( x \). If we want a relative error, then we use:

\[
\text{err}_{x,i} = \frac{|x_i - x_{i-1}|}{x_i}
\]  (31.4)

This gives us a percent error based on our current value of \( x \). This error will work unless the root is at zero.

Regardless of what choice of error we use, we have to specify a tolerance. The tolerance tells us the
maximum allowable error. We stop the iterations when the error is less than the tolerance.

### 31.3 Successive Approximations

A primitive technique to solve for the roots of \( f(x) \) is called successive approximation. In successive
approximation, we rearrange the equation so that we isolate \( x \) on left-hand side. So for the example \( f(x) \) given
below

\[
f(x) = x - \exp(-x) = 0
\]

we rearrange as:

\[
x = \exp(-x)
\]

We then make an initial guess for \( x \), plug it into the right hand side and see if it equals our guess. If it does not, we
take the new value of the right-hand side of the equation and use that for \( x \). We continue until our guess gives us
the same answer.

Successive Approximations Example 1:

Let’s find the root to the nonlinear algebraic equation given above using successive approximations. We
will use an initial guess of 0.5. We will use a relative error on \( x \) as the criterion for convergence and we will set
our tolerance at \( 10^{-6} \).

<table>
<thead>
<tr>
<th>iteration</th>
<th>( x )</th>
<th>( \exp(-x) )</th>
<th>relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5000000</td>
<td>0.6065307</td>
<td>1.0000E+02</td>
</tr>
<tr>
<td>2</td>
<td>0.6065307</td>
<td>0.5452392</td>
<td>1.1241E-01</td>
</tr>
<tr>
<td>3</td>
<td>0.5452392</td>
<td>0.5797031</td>
<td>5.9451E-02</td>
</tr>
<tr>
<td>4</td>
<td>0.5797031</td>
<td>0.5600646</td>
<td>3.5065E-02</td>
</tr>
<tr>
<td>5</td>
<td>0.5600646</td>
<td>0.5711721</td>
<td>1.9447E-02</td>
</tr>
<tr>
<td>6</td>
<td>0.5711721</td>
<td>0.5648629</td>
<td>1.1169E-02</td>
</tr>
<tr>
<td>7</td>
<td>0.5648629</td>
<td>0.5684380</td>
<td>6.2893E-03</td>
</tr>
<tr>
<td>iteration</td>
<td>x</td>
<td>-ln(x)</td>
<td>relative error</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>1</td>
<td>0.5000000</td>
<td>0.6931472</td>
<td>1.0000E+02</td>
</tr>
<tr>
<td>2</td>
<td>0.6931472</td>
<td>0.3665129</td>
<td>8.9119E-01</td>
</tr>
<tr>
<td>3</td>
<td>0.3665129</td>
<td>1.0037220</td>
<td>6.3485E-01</td>
</tr>
<tr>
<td>4</td>
<td>1.0037220</td>
<td>-0.0037146</td>
<td>2.7121E+02</td>
</tr>
<tr>
<td>5</td>
<td>-0.0037146</td>
<td>Does Not Exist</td>
<td></td>
</tr>
</tbody>
</table>

By iteration 5, we see that we are trying to take the natural log of a negative number, which does not exist. The program crashes.

This illustrates several key points about successive approximations:

<table>
<thead>
<tr>
<th>Successive Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td>• simple to understand and use</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td>• no guarantee of convergence</td>
</tr>
<tr>
<td>• very slow convergence</td>
</tr>
<tr>
<td>• need a good initial guess for convergence</td>
</tr>
</tbody>
</table>

My advice is to never use successive approximations. As a root-finding method it is completely unreliable. The only reason I present it here is to try to convince you that you should not use it.

A Matlab code which implements successive approximation is given below. Use it at your own risk.

Since I have included a code, a few comments are necessary.
The code is stored in a file called *succ_app.m*

The code is executed by typing *succ_app(x)* where *x* is the numeric value of your initial guess.

In the code we first specify the maximum number of iterations (*maxit*), the tolerance (*tol*). We initialize the error to a high value (*err*). We initialize the iteration counter (*icount*) to zero. We initialize *xold* to our initial guess. Then we have a while-loop that executes as long as *icount* is less than the maximum number of iterations and the error is greater than the tolerance. At each iteration, we evaluate *xnew* and the error. On the first iteration, we do not calculate the error, since it takes two iterations to generate a relative error.

If the code converges, it stores *xnew* as *x0* and reports that value to the user. If the maximum number of iterations is exceeded, then we have not converged and the code let’s us know that.

The right-hand side of the equation that we want to solve is given as the last line in the code.
% succesive approximations

function [x0,err] = succ_app(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold = x0;
while (err > tol & icount <= maxit)
    icount = icount + 1;
    xnew = funkeval(xold);
    if  (icount > 1)
        err = abs((xnew - xold)/xnew);
    end
    fprintf(1,'icount = %i xnew = %e xold = %e err = %e \n',icount, xnew, xold, err);
    xold = xnew;
end
x0 = xnew;
if  (icount >= maxit)
    % you ran out of iterations
    fprintf(1,'Sorry. You did not converge in %i iterations.\n',maxit);
    fprintf(1,'The final value of x was %e \n', x0);
    fprintf(1,'Maybe you should try a better method because Successive Approximations is lame. \n');
end

function x = funkeval(x0)
x = exp(-x0);
31.4 Bisection Method of Rootfinding

Another primitive method for finding roots which we will never use in practice, but which you should be exposed to once is called the bisection method. In the bisection method we still want to find the root to
\[ f(x) = 0 \] (31.1)

We do so by finding a value of \( x \), namely \( x_+ \), where \( f(x) > 0 \) and a second value of \( x \), namely \( x_- \), where \( f(x) < 0 \). These two values of \( x \) are called brackets. If we have brackets, then we know that the value of \( x \) for which \( f(x) = 0 \) lies somewhere between the two brackets.

In the bisection method, we first must find the brackets. After we have the brackets, we then find the value of \( x \) midway between the brackets.

\[ x_{\text{mid}} = \frac{x_+ + x_-}{2} \]

If \( f(x_{\text{mid}}) > 0 \), then we replace \( x_+ \) with \( x_{\text{mid}} \), namely \( x_+ = x_{\text{mid}} \). The other possibility is that \( f(x_{\text{mid}}) < 0 \), in which case \( x_- = x_{\text{mid}} \).

With our new brackets, we find the new midpoint and continue the procedure until we have reached the desired tolerance.

Bisection Method Example 1:
Let’s solve the problem that the successive approximations problem could not solve.

\[ f(x) = x + \ln(x) = 0 \]

We will take as our brackets,
\[ x_- = 0.1 \text{ where } f(x_-) = -2.203 < 0 \]
\[ x_+ = 1.0 \text{ where } f(x_+) = 1.0 > 0 \]

How did we find these brackets? Trial and error or we plot \( f(x) \) vs \( x \) and get some ideas where the function is positive and negative.

We will use a relative error on \( x \) as the criterion for convergence and we will set our tolerance at \( 10^{-6} \).

<table>
<thead>
<tr>
<th>iteration</th>
<th>( x_- )</th>
<th>( x_+ )</th>
<th>( f(x_-) )</th>
<th>( f(x_+) )</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.500000E-01</td>
<td>1.000000E+00</td>
<td>-4.783700E-02</td>
<td>1.000000E+00</td>
<td>4.500000E-01</td>
</tr>
<tr>
<td>2</td>
<td>5.500000E-01</td>
<td>7.750000E-01</td>
<td>-4.783700E-02</td>
<td>5.201078E-01</td>
<td>2.903226E-01</td>
</tr>
<tr>
<td>3</td>
<td>5.500000E-01</td>
<td>6.625000E-01</td>
<td>-4.783700E-02</td>
<td>2.507653E-01</td>
<td>1.698113E-01</td>
</tr>
<tr>
<td>4</td>
<td>5.500000E-01</td>
<td>6.062500E-01</td>
<td>-4.783700E-02</td>
<td>1.057872E-01</td>
<td>9.278351E-02</td>
</tr>
<tr>
<td>5</td>
<td>5.500000E-01</td>
<td>5.781250E-01</td>
<td>-4.783700E-02</td>
<td>3.015983E-02</td>
<td>4.864865E-02</td>
</tr>
<tr>
<td>6</td>
<td>5.640625E-01</td>
<td>5.781250E-01</td>
<td>-8.527718E-03</td>
<td>3.015983E-02</td>
<td>2.432432E-02</td>
</tr>
<tr>
<td>7</td>
<td>5.640625E-01</td>
<td>5.710938E-01</td>
<td>-8.527718E-03</td>
<td>1.089185E-02</td>
<td>1.231190E-02</td>
</tr>
<tr>
<td>8</td>
<td>5.640625E-01</td>
<td>5.675781E-01</td>
<td>-8.527718E-03</td>
<td>1.089185E-02</td>
<td>6.194081E-03</td>
</tr>
<tr>
<td>9</td>
<td>5.658203E-01</td>
<td>5.675781E-01</td>
<td>-3.658408E-03</td>
<td>1.201251E-03</td>
<td>3.097041E-03</td>
</tr>
<tr>
<td>10</td>
<td>5.666992E-01</td>
<td>5.675781E-01</td>
<td>-1.227376E-03</td>
<td>1.201251E-03</td>
<td>1.548520E-03</td>
</tr>
<tr>
<td>11</td>
<td>5.671387E-01</td>
<td>5.675781E-01</td>
<td>-1.276207E-05</td>
<td>1.201251E-03</td>
<td>7.742602E-04</td>
</tr>
<tr>
<td>12</td>
<td>5.671387E-01</td>
<td>5.673584E-01</td>
<td>-1.276207E-05</td>
<td>5.943195E-04</td>
<td>3.872800E-04</td>
</tr>
<tr>
<td>13</td>
<td>5.671387E-01</td>
<td>5.672485E-01</td>
<td>-1.276207E-05</td>
<td>2.907975E-04</td>
<td>1.936775E-04</td>
</tr>
</tbody>
</table>
So we have converged to a final answer of $x = 0.567143$.

This example illustrates several key points about successive approximations:

<table>
<thead>
<tr>
<th>Bisection Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

A Matlab code which implements bisection method is given below.

Since I have included a code, a few comments are necessary.

The code is stored in a file called `bisection.m`

The code is executed by typing `bisection(xn,xp)` where $xn$ is the numeric value of $x$ which gives a negative value of $f(x)$ and $xp$ is the numeric value of $x$ which gives a positive value of $f(x)$. You must give the code 2 arguments for it to run properly. Additionally, these two values must be legitimate brackets.

The code follows the same outline as the successive approximations code. See comments for that code.

The equation, $f(x)$, that we want to find the roots for, is given as the last line in the code.
% bisection method

function [x0,err] = bisection(xn,xp);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
fn = funkeval(xn);
fp = funkeval(xp);
while (err > tol & icount <= maxit)
    icount = icount + 1;
    xmid = (xn + xp)/2;
    fmid = funkeval(xmid);
    if (fmid > 0)
        fp = fmid;
        xp = xmid;
    else
        fn = fmid;
        xn = xmid;
    end
    err = abs((xp - xn)/xp);
    fprintf(1,'icount = %i xn = %e xp = %e fn = %e fp = %e err = %e \n',icount, xn, xp, fn, fp, err);
end

x0 = xmid;
if (icount >= maxit)

x0 = xp;
end

function f = funkeval(x)
f = x + log(x);
31.5 Single Variable Newton-Raphson - Theory

One of the most useful root-finding techniques is called the Newton-Raphson method. The Newton-Raphson technique allows you to find solutions to a problem of the form:

\[ f(x) = 0 \]  \hspace{1cm} (31.1)

where \( f(x) \) can be any function. The basis of the Newton-Raphson method lies in the fact that we can approximate the derivative of \( f(x) \) numerically.

\[ f'(x_1) = \frac{df(x_1)}{dx} \approx \frac{f(x_1) - f(x_2)}{x_1 - x_2} \]  \hspace{1cm} (31.2)

Now, initially, we don’t know the roots of \( f(x) \). Let’s say our initial guess is \( x_1 \). Let’s say that we want \( x_2 \) to be a solution to \( f(x) = 0 \). Let’s rearrange the equation to solve for \( x_2 \).

\[ x_2 = x_1 - \frac{f(x_1) - f(x_2)}{f'(x_1)} \]  \hspace{1cm} (31.3)

Now, if \( x_2 \) is a solution to \( f(x) = 0 \), then \( f(x_2) = 0 \) and the equation becomes:

\[ x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \]  \hspace{1cm} (31.4)

This is the Newton-Raphson Method. Based on the \( x_1, f(x_1), f'(x_1) \), we estimate the root to be at \( x_2 \). Of course, this is just an estimate. The root will not actually be at \( x_2 \). Therefore, we can do the Newton-Raphson Method again, a second iteration.

\[ x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} \]  \hspace{1cm} (31.5)

In fact, we repeat these iteration using the formula

\[ x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \]  \hspace{1cm} (31.5)

until the difference between \( x_{i+1} \) and \( x_i \) is small enough to satisfy us.

The Newton-Raphson method requires you to calculate the first derivative of the equation, \( f'(x) \). Sometimes this is a hassle. Additionally, we see from the equation above that when the derivative is zero, the Newton-Raphson method fails, because we divide by the derivative. This is a weakness of the method.

However because we go to the trouble to give the Newton-Raphson method the extra information about the function contained in the derivative, it will converge must faster than the previous methods. We will see this demonstrated in the following example.

Finally, as with any root-finding method, the Newton-Raphson method requires a good initial guess of the root.
Newton Raphson Example 1:
Let’s solve the problem that the successive approximations problem could not solve.

\[ f(x) = x + \ln(x) = 0 \]

The derivative is

\[ f'(x) = 1 + \frac{1}{x} \]

We will use an initial guess of 0.5. We will use a relative error on \( x \) as the criterion for convergence and we will set our tolerance at \( 10^{-6} \).

<table>
<thead>
<tr>
<th>iteration</th>
<th>( x_{old} )</th>
<th>( f(x_{old}) )</th>
<th>( f'(x_{old}) )</th>
<th>( x_{new} )</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.000000E-01</td>
<td>-1.931472E-01</td>
<td>3.000000E+00</td>
<td>5.643824E-01</td>
<td>1.000000E+02</td>
</tr>
<tr>
<td>2</td>
<td>5.643824E-01</td>
<td>-7.640861E-03</td>
<td>2.771848E+00</td>
<td>5.671390E-01</td>
<td>4.860527E-03</td>
</tr>
<tr>
<td>3</td>
<td>5.671390E-01</td>
<td>-1.189933E-05</td>
<td>2.763236E+00</td>
<td>5.671433E-01</td>
<td>7.586591E-06</td>
</tr>
<tr>
<td>4</td>
<td>5.671433E-01</td>
<td>-2.877842E-11</td>
<td>2.763223E+00</td>
<td>5.671433E-01</td>
<td>1.836358E-11</td>
</tr>
</tbody>
</table>

So we converged to 0.5671433 in only four iterations. We see that for the last three iterations, the error dropped quadratically. By that we mean

\[ \frac{\text{err}_{i+1}}{\text{err}_i^2} \approx 1 \]

Quadratic convergence is a rule of thumb. The ratio ought to be on the order of 1.

\[ \frac{\text{err}_3}{\text{err}_2^2} = \frac{7.6 \cdot 10^{-6}}{(4.9 \cdot 10^{-3})^2} = 0.32 \quad \text{and} \quad \frac{\text{err}_4}{\text{err}_3^2} = \frac{1.83 \cdot 10^{-11}}{(7.6 \cdot 10^{-6})^2} = 0.32 \]

This example illustrates several key points about successive approximations:

<table>
<thead>
<tr>
<th>Newton-Raphson Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

A Matlab code which implements Newton-Raphson method is given below.

Since I have included a code, a few comments are necessary.

The code is stored in a file called newraph.m
The code is executed by typing newraph(x) where x is the numeric value of your initial guess of x.

The code follows the same outline as the successive approximations code. See comments for that code.

The equations for \( f(x) \) and \( dfdx(x) \) are given near the bottom of the code.
function \[ x_0, \text{err} \] = newraph(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold = x0;
while (err > tol & icount <= maxit)
    icount = icount + 1;
    f = funkeval(xold);
    df = dfunkeval(xold);
    xnew = xold - f/df;
    if (icount > 1)
        err = abs((xnew - xold)/xnew);
    end
    fprintf(1, 'icount = %i xold = %e f = %e df = %e xnew = %e err = %e \n', icount, xold, f, df, xnew, err);
    xold = xnew;
end

function f = funkeval(x)
f = x + log(x);

function df = dfunkeval(x)
df = 1 + 1/x;
31.6 Single Variable Newton-Raphson - Numerical Approximation to Derivatives

People don’t like to take derivatives. Because they don’t like to, they don’t want to use the Newton-Raphson method.

\[ x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \]  \hspace{1cm} (31.5)

So, if we are one of those people, we can approximate the derivative at \( x_i \) using a centered finite difference formula:

\[ f'(x_i) = \frac{f(x_i + h) - f(x_i - h)}{2h} \]

where \( h \) is some small number. Generally I define \( h \) as

\[ h = \min(0.01 \cdot x_i, 0.01) \]

This is just a rule of thumb that I made up that seems to work 95% of the time. Using this approximation, we execute the Newton-Raphson algorithm in precisely the same way, except we never to have to evaluate the derivative.

Newton Raphson with Numerical Approximation to the Derivative Example 1:
Let’s solve the problem that the successive approximations problem could not solve.

\[ f(x) = x + \ln(x) = 0 \]

We will use an initial guess of 0.5. We will use a relative error on \( x \) as the criterion for convergence and we will set our tolerance at \( 10^{-6} \).

<table>
<thead>
<tr>
<th>iteration</th>
<th>( x_{\text{old}} )</th>
<th>( f(x_{\text{old}}) )</th>
<th>( f'(x_{\text{old}}) )</th>
<th>( x_{\text{new}} )</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.000000E-01</td>
<td>-1.931472E-01</td>
<td>3.000067E+00</td>
<td>5.643810E-01</td>
<td>1.000000E+02</td>
</tr>
<tr>
<td>2</td>
<td>5.643810E-01</td>
<td>-7.644827E-03</td>
<td>2.771912E+00</td>
<td>5.671389E-01</td>
<td>4.862939E-03</td>
</tr>
<tr>
<td>3</td>
<td>5.671389E-01</td>
<td>-1.206407E-05</td>
<td>2.763295E+00</td>
<td>5.671433E-01</td>
<td>7.697929E-06</td>
</tr>
<tr>
<td>4</td>
<td>5.671433E-01</td>
<td>-2.862443E-10</td>
<td>2.763282E+00</td>
<td>5.671433E-01</td>
<td>1.826496E-10</td>
</tr>
</tbody>
</table>

So we converged to 0.5671433 in only four iterations, just as it did in the rigorous Newton-Raphson method. This example illustrates several key points about successive approximations:

<table>
<thead>
<tr>
<th>Newton-Raphson with Numerical Approximation of the Derivative Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td>• simple to understand and use</td>
</tr>
<tr>
<td>• quadratic (fast) convergence, near the root</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td>• blows up when derivative is zero.</td>
</tr>
<tr>
<td>• need a good initial guess for convergence</td>
</tr>
</tbody>
</table>

A Matlab code which implements Newton-Raphson method is given below.

Since I have included a code, a few comments are necessary. The code is stored in a file called `newraph_nd.m`. The code is executed by typing `newraph_nd(x)` where \( x \) is the numeric value of your initial guess of \( x \).
The code follows the same outline as the Newton Raphson code. See comments for that code. The equations for \( f(x) \) and \( df/dx(x) \) are given near the bottom of the code.
% Newton-Raphson method with numerical approximations to the derivative.

function [x0,err] = newraph_nd(x0);

maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;

xold = x0;

while (err > tol & icount <= maxit)
    icount = icount + 1;
    f = funkeval(xold);
    h = min(0.01*xold,0.01);
    df = dfunkeval(xold,h);
    xnew = xold - f/df;
    if (icount > 1)
        err = abs((xnew - xold)/xnew);
    end
    fprintf(1, 'icount = %i xold = %e f = %e df = %e xnew = %e err = %e \n',
        icount, xold, f, df, xnew, err);
    xold = xnew;
end

x0 = xnew;

if (icount >= maxit)
    % you ran out of iterations
    fprintf(1,'Sorry. You did not converge in %i iterations.\n', maxit);
end

function f = funkeval(x)
f = x + log(x);

function df = dfunkeval(x,h)
fp = funkeval(x+h);
fn = funkeval(x-h);
df = (fp - fn)/(2*h);
31.7 MATLAB fzero.m - Search and Interpolate

Matlab has an intrinsic function to find the root of a single nonlinear algebraic equation. The routine is called fzero.m. You can access help on it by typing help fzero at the Matlab command line prompt. You can also access the fzero.m file itself and examine the code line by line.

I am not going to explain the code here, but I am going to outline how the technique finds a solution. The generic technique that the fzero.m function uses to find a solution is classified as a “search and interpolation” technique.

In a “search and interpolation” method, you start with an initial guess to the equation

\[ f(x) = 0 \]

You evaluate the function at the initial guess. If the function is positive, your initial guess becomes your positive bracket, just as in the bisection case. If the function is negative, your initial guess becomes your negative bracket.

Next, you start searching around the initial guess for the other bracket. The steps might be something of along the lines of

\[
x_1 = x_0 + h \\
x_2 = x_0 - h \\
x_3 = x_0 + 2h \\
x_4 = x_0 - 2h
\]

and so forth. At each step, you evaluate the function until you find a value of \( x \) which gives you a functional evaluation which is opposite in sign to that of your initial guess.

At that point, you begin a bisection-like convergence criteria, except that instead of using the midpoint as your new guess, you use a linear interpolation between the two brackets.

\[
x_{\text{new}} = x_- - \frac{f(x_-)}{f(x_-) - f(x_+)}(x_- - x_+)
\]

You replace whichever bracket ought to be replaced with \( x_{\text{new}} \), based on the sign of \( f(x_{\text{new}}) \).

You repeat this procedure until you converge to the desired tolerance.

The actual Matlab code is a little more sophisticated but we now understand the gist behind a “search and interpolate” method.

The simplest syntax for using the fzero.m code is to type at the command line prompt:

\[
>> x = \text{fzero('f(x)',x0,tol)}
\]

where \( f(x) \) is the function we want the roots of, \( x_0 \) is the initial guess, and \( tol \) is the tolerance. An example:

\[
>> x = \text{fzero('x+log(x)',0.5,1.e-6)} \\
x = 0.56714332272548
\]

<table>
<thead>
<tr>
<th>Matlab’s fzero.m (search and interpolate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Disadvantages</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
31.8 How Many Roots do nonlinear equations have and how do we find them?

When dealing with linear algebraic equations, we could determine how many roots were possible and there were always 0, 1, or an infinite number. When dealing with nonlinear equations, we have no such theory. A nonlinear equation can have 0, 1, 2… up through an infinite number of roots. There is no sure way to tell except by plotting it out. Here are a few examples of nonlinear equations with 0, 1, 2 and an infinite number of roots.

When you use any of the numerical root-finding techniques described above, you will only find one root at a time. Which root you locate depends upon your choice of method and the initial guess.
Example:
Let’s find the four roots in the range -12.0 to 0.0 for the function depicted in the plot below, using the Newton-Raphson method.

\[ f(x) = 0.001 \cdot ((x-5)^3 \cdot \sin(x+4) - x^2 + 3) \]  \hspace{1cm} (31.9)

The derivative of this function is

\[ f'(x) = 0.001 \left[ (x-5)^3 \cos(x+4) + 3(x-5)^2 \sin(x+4) - 2x \right] \]  \hspace{1cm} (31.10)

In an attempt to find all four roots, let’s start with the one near \( x = -10.0 \). Our iterations will be logged in the following table.

<table>
<thead>
<tr>
<th>iteration</th>
<th>( x_i )</th>
<th>( f(x_i) )</th>
<th>( f'(x_i) )</th>
<th>( e_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-10</td>
<td>-1.04003</td>
<td>-2.57246</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-10.4043</td>
<td>0.336356</td>
<td>-2.90107</td>
<td>-0.40429</td>
</tr>
<tr>
<td>3</td>
<td>-10.2884</td>
<td>-0.08439</td>
<td>-2.85159</td>
<td>0.115942</td>
</tr>
<tr>
<td>4</td>
<td>-10.3179</td>
<td>0.02145</td>
<td>-2.86789</td>
<td>-0.02959</td>
</tr>
<tr>
<td>5</td>
<td>-10.3105</td>
<td>-0.00541</td>
<td>-2.86401</td>
<td>0.007479</td>
</tr>
<tr>
<td>6</td>
<td>-10.3124</td>
<td>0.001366</td>
<td>-2.865</td>
<td>-0.00189</td>
</tr>
<tr>
<td>7</td>
<td>-10.3119</td>
<td>-0.00034</td>
<td>-2.86475</td>
<td>0.000477</td>
</tr>
<tr>
<td>8</td>
<td>-10.312</td>
<td>8.71E-05</td>
<td>-2.86482</td>
<td>-0.00012</td>
</tr>
<tr>
<td>9</td>
<td>-10.312</td>
<td>-2.2E-05</td>
<td>-2.8648</td>
<td>3.04E-05</td>
</tr>
</tbody>
</table>

Therefore, one of the roots of \( f(x) \) in the range between -12.0 and 0.0 is about -10.312.
The second root in the plot of $f(x)$, looks like it’s about -7.0. Let’s use the same procedure to find that root.

<table>
<thead>
<tr>
<th>iteration</th>
<th>$x_i$</th>
<th>$f(x_i)$</th>
<th>$f'(x_i)$</th>
<th>$e_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-7</td>
<td>0.197855</td>
<td>1.29703</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-7.15254</td>
<td>-0.06781</td>
<td>1.365912</td>
<td>-0.15254</td>
</tr>
<tr>
<td>3</td>
<td>-7.1029</td>
<td>0.021133</td>
<td>1.346601</td>
<td>0.049648</td>
</tr>
<tr>
<td>4</td>
<td>-7.11859</td>
<td>-0.00674</td>
<td>1.353041</td>
<td>-0.01569</td>
</tr>
<tr>
<td>5</td>
<td>-7.11361</td>
<td>0.002132</td>
<td>1.35103</td>
<td>0.004981</td>
</tr>
<tr>
<td>6</td>
<td>-7.11519</td>
<td>-0.00068</td>
<td>1.351671</td>
<td>-0.00158</td>
</tr>
<tr>
<td>7</td>
<td>-7.11469</td>
<td>0.000214</td>
<td>1.351468</td>
<td>0.0005</td>
</tr>
<tr>
<td>8</td>
<td>-7.11485</td>
<td>-6.8E-05</td>
<td>1.351532</td>
<td>-0.00016</td>
</tr>
<tr>
<td>9</td>
<td>-7.1148</td>
<td>2.15E-05</td>
<td>1.351512</td>
<td>5.03E-05</td>
</tr>
</tbody>
</table>

Therefore, one of the roots of $f(x)$ in the range between -12.0 and 0.0 is about -7.1148.

The third root in the plot of $f(x)$, looks like it’s about -4.0. Let’s use the same procedure to find that root.

<table>
<thead>
<tr>
<th>iteration</th>
<th>$x_i$</th>
<th>$f(x_i)$</th>
<th>$f'(x_i)$</th>
<th>$e_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-4</td>
<td>-0.013</td>
<td>-0.478</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-4.0272</td>
<td>0.006786</td>
<td>-0.48292</td>
<td>-0.0272</td>
</tr>
<tr>
<td>3</td>
<td>-4.01314</td>
<td>-0.00348</td>
<td>-0.48042</td>
<td>0.014052</td>
</tr>
<tr>
<td>4</td>
<td>-4.02039</td>
<td>0.001801</td>
<td>-0.48172</td>
<td>-0.00725</td>
</tr>
<tr>
<td>5</td>
<td>-4.01665</td>
<td>-0.00093</td>
<td>-0.48105</td>
<td>0.003739</td>
</tr>
<tr>
<td>6</td>
<td>-4.01858</td>
<td>0.000479</td>
<td>-0.4814</td>
<td>-0.00193</td>
</tr>
<tr>
<td>7</td>
<td>-4.01758</td>
<td>-0.00025</td>
<td>-0.48122</td>
<td>0.000995</td>
</tr>
<tr>
<td>8</td>
<td>-4.0181</td>
<td>0.000127</td>
<td>-0.48131</td>
<td>-0.00051</td>
</tr>
<tr>
<td>9</td>
<td>-4.01783</td>
<td>-6.6E-05</td>
<td>-0.48126</td>
<td>0.000265</td>
</tr>
</tbody>
</table>

Therefore, one of the roots of $f(x)$ in the range between -12.0 and 0.0 is about -4.01783.

The fourth root in the plot of $f(x)$, looks like it’s about -1.0. Let’s use the same procedure to find that root.

<table>
<thead>
<tr>
<th>iteration</th>
<th>$x_i$</th>
<th>$f(x_i)$</th>
<th>$f'(x_i)$</th>
<th>$e_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-0.02848</td>
<td>0.108919</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-0.7385</td>
<td>0.025059</td>
<td>0.09101</td>
<td>0.261496</td>
</tr>
<tr>
<td>3</td>
<td>-1.01384</td>
<td>-0.0317</td>
<td>0.109713</td>
<td>-0.27534</td>
</tr>
<tr>
<td>4</td>
<td>-0.72492</td>
<td>0.027447</td>
<td>0.08963</td>
<td>0.288929</td>
</tr>
<tr>
<td>5</td>
<td>-1.03001</td>
<td>-0.0355</td>
<td>0.110616</td>
<td>-0.3051</td>
</tr>
<tr>
<td>6</td>
<td>-0.70906</td>
<td>0.030184</td>
<td>0.088733</td>
<td>0.320948</td>
</tr>
<tr>
<td>7</td>
<td>-1.04923</td>
<td>-0.04008</td>
<td>0.111654</td>
<td>-0.34016</td>
</tr>
<tr>
<td>8</td>
<td>-0.69022</td>
<td>0.033365</td>
<td>0.087257</td>
<td>0.359008</td>
</tr>
<tr>
<td>9</td>
<td>-1.0726</td>
<td>-0.04575</td>
<td>0.112862</td>
<td>-0.38238</td>
</tr>
<tr>
<td>10</td>
<td>-0.66724</td>
<td>0.037139</td>
<td>0.08544</td>
<td>0.405353</td>
</tr>
<tr>
<td>11</td>
<td>-1.10192</td>
<td>-0.05299</td>
<td>0.114292</td>
<td>-0.43468</td>
</tr>
<tr>
<td>12</td>
<td>-0.63824</td>
<td>0.041736</td>
<td>0.083121</td>
<td>0.463676</td>
</tr>
<tr>
<td>13</td>
<td>-1.14036</td>
<td>-0.06271</td>
<td>0.116009</td>
<td>-0.50211</td>
</tr>
<tr>
<td>14</td>
<td>-0.59976</td>
<td>0.047553</td>
<td>0.080009</td>
<td>0.540601</td>
</tr>
</tbody>
</table>
Very quickly we see this doesn’t work. It has shot us off to some other root. Why is this the case? Well, from the plot of $f(x)$, we see that there is a root at about $x = 1.0$. However, if we look at the plot of $f'(x)$, we see that the derivative is close to zero there. Since the derivative is close to zero at that root, the Newton-Raphson Method cannot find it. That’s life; sometimes things work, sometimes they don’t.

We can repeat this problem in MATLAB, you can use the fzero function. The three inputs are the function, $f(x)$, the initial guess, and the tolerance. Examples using the above function:

```matlab
x = fzero('0.001*((x-5)^3*sin(x+4) - x^2 + 3)',-10,1.e-6)
x = -10.3120
```

Changing only the second input, the initial guess, we can arrive at the data in this table:

<table>
<thead>
<tr>
<th>initial guess</th>
<th>final answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-11</td>
<td>-10.3120</td>
</tr>
<tr>
<td>-10</td>
<td>-10.3120</td>
</tr>
<tr>
<td>-9</td>
<td>-10.3120</td>
</tr>
<tr>
<td>-8</td>
<td>-7.1148</td>
</tr>
<tr>
<td>-7</td>
<td>-7.1148</td>
</tr>
<tr>
<td>-6</td>
<td>-7.1148</td>
</tr>
<tr>
<td>-5</td>
<td>-4.0179</td>
</tr>
<tr>
<td>-4</td>
<td>-4.0179</td>
</tr>
<tr>
<td>-3</td>
<td>-4.0179</td>
</tr>
<tr>
<td>-2</td>
<td>-0.8695</td>
</tr>
<tr>
<td>-1</td>
<td>-0.8695</td>
</tr>
<tr>
<td>0</td>
<td>-0.8695</td>
</tr>
</tbody>
</table>

MATLAB is using a different technique than Newton-Raphson. This method can find the root near -1.0.

We can get an idea for what sort of algorithm MATLAB is using by including a fourth input in the fzero function. This gives us a trace of the fzero solution path.

```matlab
x = fzero('0.001*((x-5)^3*sin(x+4) - x^2 + 3)',-10,1.e-6,1)
```

```
Func evals  x  f(x)  Procedure
1  -10 -1.04003  initial
2  -9.71716  -1.80091  search
3  -10.2828  -0.10396  search
4  -9.6  -2.05375  search
5  -10.4  -0.19005  search
```

Looking for a zero in the interval [-9.6, -10.4]

```
6  -10.292  -0.0713853  interpolation
7  -10.317  -0.0964944  interpolation
8  -10.312  4.04806e-007  interpolation
9  -10.312 -7.35963e-005  interpolation
```

$x = -10.3120$

MATLAB scans the local area around our initial guess until it finds a change in sign for $f(x)$. Once it finds that change in sign, MATLAB uses an iterative interpolation scheme to find the solution. This scheme is based on the
method of bisection which works if you have an upper and lower bound for the solution, which you do when, you have \( f(x) \) positive at one point (\( x = 10.4 \)) and negative at another point (\( x = 10.0 \)). Then \( f(x) = 0 \), somewhere between them.

One more thing about the Newton Raphson Method. In the above example, the equation was non-linear. If the equation is linear, the NRM takes one iteration to converge to the exact solution. For example,

\[
\begin{align*}
  f(x) &= 3x + 2 = 0 \quad \text{(31.11)} \\
  f'(x) &= 3 \\ \\
  \text{Let's have an initial guess of 0.} \\
  \text{iteration} & \quad x_i & \quad f(x_i) & \quad f'(x_i) & \quad e_i \\
  1 & \quad 0 & \quad 2 & \quad 3 & \\
  2 & \quad -0.66667 & \quad 0 & \quad 3 & \quad -0.66667 \\
  3 & \quad -0.66667 & \quad 0 & \quad 3 & \quad 0 \\
\end{align*}
\]

The first value of \( x \) after our initial guess is the exact answer. The NRM solves linear equations exactly in one iteration.
31.9 Single Variable Rootfinding - MATLAB - rootfinder.m

On the website, there is a code called rootfinder.m, written by Dr. Keffer at UTK. This code will combine the rootfinding techniques of the intrinsic MATLAB function, fzero, with the plotting capabilities of MATLAB.

The description for how to use the file can be obtained by opening MATLAB, moving to the directory where you have downloaded the rootfinder.m file, and typing

```
help rootfinder
```

This yields:

```
rootfinder(xo,xstart,xend)
rootfinder will find the root of a single non-linear algebraic equation, given the initial estimate, xo.
It will plot the function in the range xstart to xend.
The function must be given in the file fnctn.m
Any and all of the arguments to rootfinder are optional.

Author: David Keffer Date: October 22, 1998
```

So, for example, if you entered, at the MATLAB command line interface,

```
rootfinder(1,0,10)
```

this would find the root nearest your initial guess of 1 and then plot the function from 0 to 10, putting markers at the point of your initial guess as well as at the converged root. If you omit the limits of plotting, then the program chooses some (perhaps unhelpful) default limits.

The equation to be solved by rootfinder must be located in the file fnctn.m. It can be as simple as:

```
function f = fnctn(x)
f = 0.001*((x/4-5)^3*sin(x/4+4) - (x^2)/16 + 3)*exp(-x/40);
```

Or, the file fnctn.m can be more complicated, such as finding the roots of the van der Waal’s equation of state.

```
function f = fnctn(x)
% van der Waal's equation of state
a=0.1381; % m^6/mol^2
b=3.184*10^-5; % m^3/mol
R=8.314; % J/mol/K
T=98; % K
P = 101325; % Pa
f = R*T/(x-b)-a/x^2 - P;
```

It doesn’t matter how complicated the fnctn.m file is so long as, at the end, you give the value of the function of interest.

An example:
Let’s find the three roots (the vapor, liquid, and intermediate molar volume) of the van der Waal’s equation of state. The fnctn.m file is given above.

First we find the vapor root. We can get an initial guess for this value from the ideal gas law.

\[
\frac{V}{n} = \frac{RT}{P} = \frac{8.314 \cdot 98}{101325} = 0.00804
\]
>> rootfinder(0.008,1.0e-3,1.0e-1)
an = 0.00790121184303

Here the second and third arguments are just lower and upper limits on the x-axis of the plot. In the plot, the blue square is the initial guess and the red star is the root.

Second, let’s find the liquid root, which is probably about 20% larger than the parameter b

```matlab
» b = 3.184*10^-5;
» rootfinder(1.2*b,1.1*b,1.0e-4)
an = 4.246507385109214e-005
```

Last, let’s find the intermediate roots, which is somewhere between the two roots we have already found.

```matlab
>> rootfinder(1.0e-3,1.0e-4,1.0e-3)
an = 1.293375218680640e-004
```
Having the plots available helps us locate good initial guesses for the three roots to van der Waal’s equation of state.
31.10 Single Variable Rootfinding - MATLAB - GUIs

A GUI (pronounced gooey) is a graphical user interface, which makes using codes simpler. There is a GUI for rootfinding, developed by Dr. Keffer, located at: http://clausius.engr.utk.edu/webresource/index.html. You have to download and unzip the GUI. Then, when you are in the directory where you extracted the files, you type:

```matlab
>>aesolver1_gui
```

at the command line prompt to start the GUI.

This GUI, titled aesolver1_gui, solves 1 algebraic equation and provides a plot. The purpose of the GUI is to be self-explanatory. I just make a few comments here. The equation to be solved can either be into this GUI directly. You can choose one of four methods to find the root

- `fzero.m` (search and interpolate)
- Newton Raphson with Numerical Approximations to the Derivative
- Brent’s Line Minimization without derivatives
- Brent’s Line Minimization with derivatives

These last two methods are taken from Numerical Recipes, the best existing resource for numerical methods. This book happens to be available on-line for free at http://www.ulib.org/webRoot/Books/Numerical_Recipes/. If you are interested about Brent’s line minimization, you can read about it in their book.

Brent’s Line Minimization is a minimization routine, not a root-finding technique. A root-finding technique solves

\[ f(x) = 0 \]

A minimization technique simply finds a local minima of \( f(x) \). Generally we think of a local minima as being defined as the \( x \) where

\[ f'(x) = 0 \]

However in these minimization methods, they are bracketing routines (like Bisection) that search for the lowest value of \( f(x) \) that they can find.

A minimization routine can be used to find the root of a function, if you take the absolute value of the function.

An example: Consider the function

\[ f(x) = x + \ln(x) \]

The function and its absolute value are plotted below. Root-finding techniques use the function itself to find the root. A minimization routine uses the absolute value of the function to find the smallest value of the function, which in a root-finding problem is always zero.

The GUI offers multiple methods because each method has strengths and weaknesses. If one of the methods doesn’t work, try another one. If none of them work, try an initial guess.
Root-finding on the plot on the left or minimization of the plot on the right, yields the same result, namely the root is located at 0.567154.

\[ f(x) = x + \ln(x) \]

\[ f(x) = |x + \ln(x)| \]